A Design for a Multimedia Server using QuickTime

by

Conan Brian Dailey

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1995

© Conan Brian Dailey, MCMXCV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author .....
Department of Electrical Engineering and Computer Science
January 13, 1995

Certified by .....
William J. Qualls
Associate Professor
Thesis Supervisor

Accepted by .....
F. R. Morgenthaler
Chairman, Departmental Committee on Undergraduate Theses

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

AUG 10 1995

Barker Eng

A Design for a Multimedia Server using QuickTime

by

Conan Brian Dailey

Submitted to the Department of Electrical Engineering and Computer Science
on January 13, 1995, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

A multimedia server was developed using the facilities of the MacOS and QuickTime libraries. It allows multiple clients to simultaneously access QuickTime movies and improves significantly on the service provided by the built-in AppleShare file server in the context of multimedia. The parameters that guided the development of the server are presented as are the specific ways in which they limit the performance of the server. Results to performance tests run on both server models are compared indicating the superiority of the new multimedia server in relation to the AppleShare server.

Thesis Supervisor: William J. Qualls
Title: Associate Professor

Acknowledgments

I would first like to thank my Mom and Dad for their unwavering support and the prodding when necessary. I would also like to express my gratitude for my wife Mireille (a.k.a. B.C.) without whom this would not have been possible. Finally, I would like to thank Bill Qualls, Glen Urban, and Jon Bohlmann for their faith in my ideas.

Contents

List of Tables

Chapter 1

Introduction

Most of the literature which discusses multimedia servers refers to the problem of providing this service as some sort of "continuous" process perhaps because the analog counterparts to the types of services provided—typically video and audio signals—require a continuous signal in that domain. However, by their very nature, digital processes are not continuous and referring to them as such creates an important problem: by referring to the process as continuous it biases the way the systems are implemented in that designers try to provide a continuous service where no need exists. In fact, a continuous service is of no real use to humans anyway—hearing and vision both depend on a time-averaged sample of the data they sense. The granularity of hearing is about one-fiftieth of a millisecond and the granularity of vision is about one-thirtieth of a second. On a human timescale these may certainly seem continuous, but in the arena of electronic transmission, where transfer speeds approach the speed of light, these intervals represent eons. Therefore, it is instructive to treat the multimedia server problem just as one would treat any other data transfer problem which has certain bandwidth and time requirements for its data.

More interesting than a treatise on what a multimedia server should be called though is why it should be built in the first place—there are two main reasons. First, it is economical: if you have a network in place that will support the bandwidth requirements or if the cost of installing a network is cheaper than the cost of a multigigabyte drive at each client, installing a server only makes sense. In fact, the

cost per byte of network bandwidth is dropping by a factor of ten every eighteen months, around three times as fast as the cost per byte of disk is dropping, so even if it is not economical now, it will be soon. Second, maintenance is much easier at one or a few locations than at every client.

So a multimedia server is a good idea, but why is it the subject of a thesis project? The impetus for designing and implementing one sprang from failed attempts to use Macintosh computers as multimedia servers relying solely on the services provided by AppleShare, the file server built into the MacOS. Therefore the primary research question for this thesis is how can a better multimedia server be built using QuickTime, bypassing the AppleShare server which performs poorly in the context of multimedia.

First a summary of work that has been done which relates to multimedia servers is presented. Then the aforementioned problems with the AppleShare multimedia server are explained. Next, the parameters which govern the performance of a multimedia server will be explored and an equation relating them developed. Thereafter, the setup of the server's software will be documented to be followed by a discussion of the testing that was done to arrive at values for the performance parameters. Next, the actual design of the server will be explored followed by a look at its performance using the same benchmarks the AppleShare server was tested against. Finally, some suggestions will be made for future improvements.

Chapter 2

Past approaches

First of all, what is a multimedia server? By taking a look at the literature on the subject we can find two main interpretations to this question. To some[1, 2, 6], a multimedia server encompasses both input and output of audio and video data consisting of both dynamic data (say from a telephony application) and static data (a digital movie stored on a disk.) Whereas others[5] have perceived the problem as solely the retrieval of audio and video data from storage. The latter is the model followed in this paper.

More varied and interesting than the definitions of a multimedia server are the issues that have been researched regarding them. Because multimedia data has strict time and bandwidth requirements some[1] have attacked the problem of scheduling resources to guarantee that these requirements will be met. Others[4] contend that for some applications it is more acceptable to degrade performance gradually in the light of bandwidth limitations instead of denying access. Another important issue is synchronization—ensuring that multiple data sources, possibly stored on different media, will be viewed properly in their temporal relationship to one another at the client. Anderson et al.[2] develop a method which utilizes a logical time system concept to support synchronization. Another popular standard for multimedia applications on the Macintosh platform, QuickTime, uses a similar method based on a time coordinate system to provide synchronization.

In addition to the above problems researchers have looked at OS mechanisms for

supporting multimedia[5], storage formats for different multimedia types[3], and the support of clients who wish to share the same source of multimedia data[6].

Despite the similarity of issues faced in this research as compared to these past approaches, this research differs in a way which makes comparisons to the previous work difficult—in the above research the designers first had ideas about servers they wanted to implement and then they chose appropriate hardware and software platforms, whereas the following research already had a hardware and software platform to which the server had to conform.

Chapter 3

Problems using AppleShare

In testing the performance of QuickTime over AppleShare two main configurations of the server were used with either one or two clients. In the first trial a single QuickTime movie was played from the server at one of the clients which resulted in poor performance—two clients produced even worse performance. Problems with just one client included a frame rate decrease of fifty percent, losses in audio information, and audio/video synchronization problems. With two clients it exhibited more severe symptoms of the same problem—each client would alternate freezing up for a number of seconds and then bursting through the information for the same period of time rendering the movie incomprehensible.

However, in the second trial some of the server's main memory was used as a RAM disk. With one client this scheme produced results rivaling the performance of a client playing a movie stored at the client which is the performance benchmark. Once again though surprisingly, performance with two clients was just as poor as the first trial with two clients. These qualitative results have hinted at two problems with the server: first, that it does not naturally buffer the data it is sending to the client; and second, it does not respond efficiently to multiple requests.

Chapter 4

Parameters affecting the design of the server

What actions must be performed in order to get a movie from the server to be displayed at the client? In order to retrieve the movie the computer must first access the disk. It must then send that particular block of data over the network. After the data arrives at the client it might have to be processed, typically decompressed, before it can finally be output. These actions define the parameters that must be investigated in order to develop the server.

The setup uses three stock Power Macintosh 8100/80 AV's which come with an 80MHz PowerPC processor, 500 megabyte hard drive, built-in ethernet port, and 16 megabytes of main memory. They run the MacOS 7.5 operating system which consumes about half of the available memory.

These machines also have 256 kilobyte level 2 processor cache and a sophisticated subsystem for handling direct memory access(DMA) and other I/O on the motherboard. Each component plays some role in the performance of the computer; however, some of these items affect the server or client to a greater extent and are briefly explained below followed by equations which describe the performance limitations implied by the variables.

4.1 Disk seek time(ST) an disk transfer rate(XR)

Associated with disk access are both the seek time necessary to find the data on the disk and the transfer time necessary to move the data into RAM. The disk used in the setup is a Quantum drive mechanism whose specifications indicate an average seek time of 11.7 milliseconds and a transfer rate of 1.9 megabytes per second.

4.2 Movie data rates(DR) and the compressor

The movies are compressed with the Apple Video Compressor that Apple provides in its QuickTime extension. Using this compressor compression ratios of around 5:1 are achieved and the data rates for our movies range between 2.5 and 2.8 megabits a second in compressed form. Movie duration is hereafter represented by D.

4.3 Video decompression rate

The video decompression rate for movies is at least twice the movie rate. This was confirmed by testing to be described below and supports the belief that any bottleneck that may exist resides either at the server or the network.

4.4 Network bandwidth/realized bandwidth(B)

The standard bandwidth ascribed to an Ethernet network is ten megabits per second. In testing also to be described below a data rate of eight megabits per second emanating from the server was realized and upwards of three megabits per second was received at the client without straining the computer.

4.5 Performance relations affecting server design

When data is read from the disk it is not retrieved continuously, it comes in frames of various sizes. Therefore for a movie stored on disk the movie data rate may be

equivalently separated into the product of the size of the frame retrieved(FS) and the frame rate(FR)

$$DR = FS \cdot FR.$$

In this way one can determine how much time the disk will take to transfer the frame

$$time = ST + \frac{FS}{XR}.$$

However, if this quantity is greater than the duration the frame represents, the server will have to buffer an amount equal to

$$DR \cdot \left(ST + \frac{FS}{XR} - \frac{1}{FR}\right)(D \cdot FR)$$

otherwise no buffering will be necessary. For servers with N clients the buffering needs increase to

$$DR \cdot \left(ST + \frac{FS}{XR} - \frac{1}{FR \cdot N}\right)(D \cdot FR)$$

if

$$time >= \frac{1}{N \cdot FR}$$

For the configuration outlined above a server running on one of the computers could serve at most three clients.

$$\left(0.0117 + \frac{35kB}{1900\frac{kB}{sec}}\right) - \frac{1}{3} \cdot \frac{10}{sec} < 0 < \left(0.0117 + \frac{35kB}{1900\frac{kB}{sec}}\right) - \frac{1}{4} \cdot \frac{10}{sec}$$

Another important and more obvious relation is that the number of clients that can use the server simultaneously is

$$N <= \frac{B}{DR}.$$

This parameter also limits the server to three clients

$$3 < \frac{8.1\frac{Mb}{sec}}{2.6\frac{Mb}{sec}} < 4$$

on the above configuration.

Chapter 5

Software overview

The Macintosh provides a complete set of system software and extensions for handling audio, video, and other forms of data. In addition to these facilities for handling data, the MacOS also administers timing services, network services, and process control services ideal for a multimedia server.

5.1 QuickTime overview

The QuickTime software libraries provide a mechanism for dealing with time-based data. This includes sound, video, animation, as well as, charts of quarterly earnings. QuickTime provides tools for capturing, storing, accessing, viewing and editing this information.

5.1.1 Structure of a movie

Regardless of the type of data QuickTime is working with that data is abstractly referred to as a movie. Each movie is composed of a number of tracks which may be based on different time scales. For instance, it probably does not make too much sense for sound which is sampled at 22kHz to use the same base time unit as quarterly sales data even though these two types of data may be stored in the same movie for presentation at a later time. In turn, each of these tracks contains a media which

contains the actual samples of the data, a description of the data, and hints for how the data should be accessed.

5.1.2 Movie toolbox

The Movie Toolbox provides the main functionality for accessing movies as well as a timing service. Specifically, the toolbox permits access to all levels of the movie data structure from the movie down to the sample. It also allows you to either specify the number of samples you would like to retrieve at a certain time or it will provide a suitable amount based on the type of media being accessed.

The timing service allows a callback procedure to be invoked at a time specified by the application. The time is calculated from a time base which specifies the time as some number of units with respect to a certain time scale, typically the time scale associated with the media the application is interested in displaying.

5.1.3 Image Compression Manager

QuickTime also provides routines for compressing and decompressing sequences of images although they work independently of the time scale with which the data is associated (if it is associated with a time scale at all.) Despite its independence from time, though Image Compression Manager still provides mechanisms which enable the data to be displayed at a precise time: when decompressing, the application can inform the routine to decompress into an offscreen image buffer to be later displayed at a precise moment. In this way the latency and unpredictability of the decompression process can be effectively masked.

5.2 MacTCP

MacTCP supplies protocols for implementing connection-based reliable packet communication in the form of TCP connections as well as unreliable packet communication in the form of UDP datagrams. It has a robust interface providing asynchronous

notification of incoming packets and allowing the asynchronous or synchronous, transmission or reception, of datagrams. The means of signaling when an asynchronous routine has completed is through the use of a callback procedure as well as by setting a state variable. The maximum packet size allowed by MacTCP is 8192 bytes. MacTCP also buffers a number of incoming packets even if there are not outstanding read operations pending. This prevents data loss in situations where the incoming data is bursty.

5.3 Thread Manager

The Thread Manager is a software extension to the MacOS which provides the ability to execute and schedule light weight processes within the context of a single application. On the PowerMac the Thread Manager which runs native PowerPC assembly code (as opposed to emulated 68K instructions) executes in a model where the current thread must explicitly yield in order to effect a context switch. In contrast with preemptive threads from which most scheduling details are abstracted, it is important that threads executing in this model are mindful of the processing needs of other threads and yield as often as possible.

In addition to the low overhead imposed on context switches, the thread mechanism is useful for memory management. Because most asynchronous operations terminate by invoking a callback procedure which occurs at interrupt time, many memory management calls are not available. However, threads which are suspended before entering the callback procedure can be asynchronously resumed at this time through an interrupt safe procedure call. Consequently, when memory is allocated a thread can be created in the suspended state which will deallocate the memory when resumed. A pointer to the thread is then passed to the callback procedure and when the callback executes it can resume the thread to deallocate the memory.

5.4 Sound Manager

The Sound Manager controls the processing of all sound within the computer and also provides a rich interface with which applications can tap its abilities. One of the main options available for sound output is through the use of sound channels. Sound channels allow the application to install buffers of sound to be played, to pause processing of the channel, and to perform a variety of other tasks related to sound management. One of the most useful features of the Sound manager is that a callback procedure can be associated with a sound channel so when a callback command is processed by the channel the procedure is called. Therefore if a callback command is issued after every play buffer command the application knows when the data is no longer needed.

5.5 Time Manager

The Time Manager provides timing services related to those provided by the Movie Toolbox but in a more flexible fashion. Instead of a procedure being called when a timebase reaches some time value as in the Movie Toolbox, the Time Manager allows a procedure to be called after an application–specified delay has elapsed. Freed from the timebase samples can be synchronized from media which have different timescales with ease; whereas using the movie toolbox, the timebase controlling one set of callbacks must slave off the other timebase which is less intuitive and more difficult to implement.

Chapter 6

Testing

In order to establish values for some of the important parameters needed to build the server, several test programs were written. First, a new client program was developed to see if the problems surrounding the AppleShare server were insurmountable. Because the assumption was that AppleShare was not buffering properly it was believed that if the client could prefetch (ask for data it didn't need right away but would need in the near future) and buffer the data at the client end a server might not have to be developed after all. Initial tests with one client were promising as data rates were high and frames synchronized; however, the AppleShare server was plagued by the same problems realized with other clients when two prefetching clients were used, which made it clear that a server had to be built.

Next, even though the theoretical bandwidth of the Ethernet is known to be ten megabits per second, confirmation was desired that the computer indeed could get data out to the network that quickly and that it could be processed fast enough at the client as well. To help acquire this information a server was developed which sent static data across the network to two clients as fast as it could using synchronous MacTCP write calls. The clients used synchronous MacTCP read calls to collect the data. The tests showed that the server could send in excess of eight megabits per second of data and the clients could receive in excess of three megabits per second. These tests clearly show that the network performance is adequate to support a multimedia server of the movies whose data rates were described above. This is

especially evident because asynchronous operations yield higher performance than synchronous operations as a general rule.

A final test program was developed where a client and server were logically on the same machine (although no data was transmitted via the network) to verify that an application could retrieve and display the samples of a movie as fast as the high level QuickTime functions did transparently. In these tests the program was able to retrieve samples, decompress them, and display them at twice the movie rate—clearly showing that a server should be able to support at least two clients without buffering, and possibly more if the bottleneck is not disk access time.

Chapter 7

Server implementation

Due to the results generated by the last test above and plans to service only two clients initially, it was decided that buffering was not necessary, therefore the server program is relatively straightforward. It was also decided that UDP datagrams would be used because their interface is simpler and the risks of data corruption on a passive hub were sufficiently low.

7.1 Main thread

The main thread first opens MacTCP port number one for sending movies and receiving requests. Next it initializes the scheduling thread and the two movie threads and yields to the scheduling thread, yielding continuously until the program is terminated at which time it deallocates memory given to MacTCP and terminates.

7.2 Scheduling thread

The scheduling thread essentially waits for requests asynchronously and passes a request to an unscheduled movie thread if one exists.

7.3 Movie thread

The movie thread performs the bulk of the work associated with the movie. It first gains access to the movie requested by the client and initializes data structures with the various tracks, media, and sample description headers for the movie. Next, it sends the sample description information to the client so it can prepare to display the movie. Finally, the thread grabs samples and send them to the client using the time manager to synchronize when samples are sent so the client is neither being choked by too much data nor being starved by a lack thereof.

7.4 Protocol

There is a very simple interface between the client and the server. For a client to receive a movie it need only send the server an index (which has a predetermined meaning at the server) indicating what movie it wants and the locations of its sound and video ports. In turn the server responds by sending sample description information to the client for the purpose of initializing data structures at the client. Finally, after a suitable delay has elapsed, during which time the client has initialized itself, the server sends sound data to the sound port and video data to the data port until the movie has played in entirety.

Chapter 8

Results

After the server was built it was tested with a number of clients conforming to the protocol specified above with achieving uniform results with the different clients. Although results were still good with one client, serving two clients still posed problems.

8.1 Better results than with AppleShare

In head to head comparisons between the AppleShare server and the custom built server using QuickTime the custom built server always fared better. Frame rates were higher, audio quality was better, and synchronization was much closer. These improvements in quality are due to better management of the resources the servers control.

Table 8.1: Performance with different servers

	AppleShare		Custom Server	
# of clients	1	2	1	2
Frame Rate	half	poor	full	poor
Audio	choppy	poor	good	good
Synchronization	ok	poor	good	poor

8.2 Explanation of lower than expected performance

Still disappointing though were video and synchronization problems exhibited in the two client test. Video seemed to be stalled from the server; however, the audio was still perfect. This type of problem calls into question the code used to synchronize the sending of the data. Essentially, if the next frame of data is not ready to be sent the Time Manager synchronization procedure executes almost continually, hogging the processor bandwidth and causing the video to stall. This should be remediable simply by altering the behavior of the synchronization procedure when the server is not quite ready to send.

Chapter 9

Suggested improvements

Several improvements could be made to the server design presented above. First, the protocol between the client and server does not allow for an arbitrary movie to be requested–only an index is passed from the client to the server which has movie references "hardcoded" into it. A more robust interface for choosing which movie the server delivers is definitely necessary for improving the server.

Secondly, the movie data rates are much too high. Therefore, a better compressor should be used which gets at least a 10:1 compression ratio. This would not only decrease the bandwidth consumed by a movie but allow more clients to interface with the server at any one time.

Finally, the options supplied by the server could be expanded. Right now the server can only send a movie—it cannot even be stopped! It would be nice to have a server which could pause, resume, rewind, and abort the movie. These are typical functionalities provided by QuickTime and they should be extended to this server.

Chapter 10

Conclusion

In the course of the above research a multimedia server was built utilizing the QuickTime multimedia library. Its performance, although not optimal, improved greatly on the status quo and the research undertaken in its development has made clear the path to further improvements. By continuing to demand a higher level of performance from computers and software the latent abilities in these systems will be realized.

Appendix A

Server Source Code

```
#include <Movies.h>
#include <Memory.h>
#include <QuickTimeComponents.h>
#include <ImageCompression.h>
#include <Sound.h>
#include <Timer.h>
#include <LSimpleThread.h>
#include <UDPPB.h>

pascal void SyncProc(TMTaskPtr task);                                    10
pascal void SyncProc2(TMTaskPtr task);
void MovieProc(LThread &thread,void *arg);
void ScheduleProc(LThread &thread,void *arg);
void schedule_complete(struct UDPiopb *iopb);

void schedule_complete(struct UDPiopb *iopb)
{
LThread::ThreadAsynchronousResume((LThread*)iopb->csParam.receive.userDataPtr);
}
                                                                         20
typedef struct {
        short                   me;
        short                   movie;
```

```
        ip_addr                 addr;
        unsigned short  vPort;
        unsigned short  sPort;
        StreamPtr               udpStream;
        short                   myRefNum;
        } MovieData;

typedef struct {
        short                   movie;
        unsigned short  vPort;
        unsigned short  sPort;
        } RequestData;

typedef struct {
        unsigned short  length;
        Ptr                             ptr;
        } WDS;

typedef struct {
        TMTask                  task;
        long                    A5world;
        } MyTask;

long duration,duration2,delay,delay2,overtime,overtime2,dtime,dtime2;
Boolean vready = false,sready = false;
const short numberOfMovies = 3;
Boolean ready[numberOfMovies];
LSimpleThread *movieThread[numberOfMovies];

pascal void SyncProc(TMTaskPtr task)
        {
        MyTask *mytask = (MyTask*)task;
        long oldA5 = SetA5(mytask->A5world);
        if (!vready) {
                delay += 2000;
                PrimeTime((QElemPtr)task,-2000);
```

```
            }
        else {
            if (duration) {
                overtime = duration + delay + overtime;
                if (overtime <= 0) {
                    PrimeTime((QElemPtr)task,overtime);
                    overtime = 0;
                    }
                else {
                    PrimeTime((QElemPtr)task,0);
                    }
                delay = 0;
                }
            vready = false;
            }
        SetA5(oldA5);
        }

pascal void SyncProc2(TMTaskPtr task)
        {
        MyTask *mytask = (MyTask*)task;
        long oldA5 = SetA5(mytask->A5world);
        if (!sready) {
            delay2 += 2000;
            PrimeTime((QElemPtr)task,-2000);
            }
        else {
            if (duration2) {
                overtime2 = duration2 + delay2 + overtime2;
                if (overtime2 <= 0) {
                    PrimeTime((QElemPtr)task,overtime2);
                    overtime2 = 0;
                    }
                else {
                    PrimeTime((QElemPtr)task,0);
                    }
```

```
                    delay2 = 0;
                    }
                sready = false;
                }
        SetA5(oldA5);                                                    100
        }

void main(void)
{
MaxApplZone();
InitGraf(&qd.thePort);
InitWindows();

short myRefNum;
OSErr e = OpenDriver("\p.IPP",&myRefNum);                                110

UDPiopb moviePort;
const short MoviePortBufLen = 200000;
moviePort.ioCRefNum = myRefNum;
moviePort.csCode = UDPCreate;
Ptr movptr = NewPtr(MoviePortBufLen);
moviePort.csParam.create.rcvBuff = (Ptr)movptr;
moviePort.csParam.create.rcvBuffLen = MoviePortBufLen;
moviePort.csParam.create.notifyProc = 0;
moviePort.csParam.create.localPort = 1;                                  120
e = PBControlSync((ParmBlkPtr)&moviePort);

MovieData movieData;
movieData.udpStream = moviePort.udpStream;
movieData.myRefNum = myRefNum;

UMainThread *mainthread = new UMainThread();
LSimpleThread *scheduleThread = new LSimpleThread(&ScheduleProc,&movieData);

for(int j=0;j<numberOfMovies;j++) movieThread[j] =                       130
                new LSimpleThread(&MovieProc,&movieData);
```

```
EnterMovies();

for(j=0;j<numberOfMovies;j++) ready[j]=true;
scheduleThread->Resume();

RgnHandle rgn = NewRgn();
EventRecord theEvent;
KeyMap keymap;                                                          140
while (!(Button() && (GetKeys(keymap),keymap[1] == 0x00000001))) {
        LThread::Yield();
        }
scheduleThread->DeleteThread();
for(j=0;j<numberOfMovies;j++) movieThread[j]->DeleteThread();
moviePort.csCode = UDPRelease;
PBControlSync((ParmBlkPtr)&moviePort);
DisposePtr(movptr);
}
                                                                        150
void ScheduleProc(LThread &thread,void *arg)
{
MovieData *moviedata = (MovieData*)arg;

UDPIOCompletionUPP schedule_comp = NewUDPIOCompletionProc(schedule_complete);

UDPiopb request;
request.udpStream = moviedata->udpStream;
request.ioCRefNum = moviedata->myRefNum;
request.csParam.receive.timeOut = 0;                                    160
request.csParam.receive.secondTimeStamp = 0;

while(true) {
        request.csCode = UDPRead;
        request.csParam.receive.userDataPtr = (Ptr)&thread;
        request.ioCompletion = (UDPIOCompletionProc)schedule_comp;
        OSErr e = PBControlAsync((ParmBlkPtr)&request);
```

```
        thread.Suspend();
        Boolean done = false;
        int j = 0;                                                          170
        while(!done && j < numberOfMovies) {
                if (ready[j]) {
                        RequestData *req = (RequestData*)request.csParam.receive.rcvBuff;
                        moviedata->addr = request.csParam.receive.remoteHost;
                        moviedata->movie = req->movie;
                        moviedata->vPort = req->vPort;
                        moviedata->sPort = req->sPort;
                        moviedata->me = j;
                        ready[j] = false;
                        movieThread[j]->Resume();                           180
                        done = true;
                        }
                else j++;
                }
        request.csCode = UDPBfrReturn;
        request.csParam.receive.userDataPtr = (Ptr)&thread;
        request.ioCompletion = (UDPIOCompletionProc)schedule_comp;
        PBControlAsync((ParmBlkPtr)&request);
        thread.Suspend();
        }                                                                   190
}

void MovieProc(LThread &thread,void *arg)
{
MovieData *moviedata = (MovieData*)arg;
short me = moviedata->me;
FSSpec spec;
Movie schindler;
OSErr e;
short resRefNum;                                                            200
UDPIOCompletionUPP schedule_comp = NewUDPIOCompletionProc(schedule_complete);
while(true) {
        if (moviedata->movie)
```

```
        e = FSMakeFSSpec(0,0,"\pMacintosh HD:phys_price",&spec);
else
        e = FSMakeFSSpec(0,0,"\pMacintosh HD:sales_capabilities",&spec);
e = OpenMovieFile(&spec,&resRefNum,fsRdPerm);
e = NewMovieFromFile(&schindler,resRefNum,nil,nil,
                        newMovieDontAskUnresolvedDataRefs,nil);
                                                                    210
Media soundMedia,videoMedia;
Track soundTrack,videoTrack;
TimeScale soundTimeScale,videoTimeScale;
SampleDescriptionHandle VsampDescH = (SampleDescriptionHandle)NewHandle(0);
SampleDescriptionHandle SsampDescH = (SampleDescriptionHandle)NewHandle(0);
OSType mediaType;
for(int j=0;j<2;j++) {
        GetMediaHandlerDescription(GetTrackMedia
                        (GetMovieIndTrack(schindler,j+1)),&mediaType,nil,nil);
        e = GetMoviesError();                                       220
        if (mediaType == 'vide') {
                videoTrack = GetMovieIndTrack(schindler,j+1);
                videoMedia = GetTrackMedia(videoTrack);
                videoTimeScale = GetMediaTimeScale(videoMedia);
                GetMediaSampleDescription(videoMedia,1,VsampDescH);
                }
        else {
                soundTrack = GetMovieIndTrack(schindler,j+1);
                soundMedia = GetTrackMedia(soundTrack);
                soundTimeScale = GetMediaTimeScale(soundMedia);     230
                GetMediaSampleDescription(soundMedia,1,SsampDescH);
                }
        }
TimeValue endTime = GetMovieDuration(schindler);

HLock((Handle)VsampDescH);
HLock((Handle)SsampDescH);
```

```
WDS IDS[3];                                                          240
IDS[0].length = (unsigned short)(**VsampDescH).descSize;
IDS[0].ptr = (Ptr)*VsampDescH;
IDS[1].length = 0;

UDPiopb initpb;
initpb.ioCRefNum = moviedata->myRefNum;
initpb.udpStream = moviedata->udpStream;
initpb.csCode = UDPWrite;
initpb.csParam.send.reserved = 0;
initpb.csParam.send.remoteHost = moviedata->addr;                    250
initpb.csParam.send.remotePort = moviedata->vPort;
initpb.csParam.send.wdsPtr = (Ptr)IDS;
initpb.csParam.send.checkSum = 0;
initpb.csParam.send.sendLength = 0;
initpb.csParam.send.userDataPtr = 0;
e = PBControlSync((ParmBlkPtr)&initpb);

IDS[0].length = (unsigned short)(**SsampDescH).descSize;
IDS[0].ptr = (Ptr)*SsampDescH;
IDS[1].length = 0;                                                   260

e = PBControlSync((ParmBlkPtr)&initpb);


if ((Handle)SsampDescH) DisposeHandle((Handle)SsampDescH);
if ((Handle)VsampDescH) DisposeHandle((Handle)VsampDescH);

WDS VDS[3];
VDS[0].length = 0;
VDS[0].ptr = nil;                                                    270
VDS[1].length = 0;
VDS[1].ptr = nil;
VDS[2].length = 0;

UDPiopb videopb;
```

```
videopb.ioCRefNum = moviedata->myRefNum;
videopb.udpStream = moviedata->udpStream;
videopb.csCode = UDPWrite;
videopb.csParam.send.reserved = 0;
videopb.csParam.send.remoteHost = moviedata->addr;                    280
videopb.csParam.send.remotePort = moviedata->vPort;
videopb.csParam.send.wdsPtr = (Ptr)VDS;
videopb.csParam.send.checkSum = 0;
videopb.csParam.send.sendLength = 0;
videopb.csParam.send.userDataPtr = (Ptr)&thread;
videopb.ioCompletion = (UDPIOCompletionProc)schedule_comp;

UDPiopb video[5];
for(int i=0;i<5;i++){
video[i].ioCRefNum = moviedata->myRefNum;                             290
video[i].udpStream = moviedata->udpStream;
video[i].csCode = UDPWrite;
video[i].csParam.send.reserved = 0;
video[i].csParam.send.remoteHost = moviedata->addr;
video[i].csParam.send.remotePort = moviedata->vPort;
video[i].csParam.send.wdsPtr = (Ptr)VDS;
video[i].csParam.send.checkSum = 0;
video[i].csParam.send.sendLength = 0;
video[i].csParam.send.userDataPtr = (Ptr)&thread;
video[i].ioCompletion = (UDPIOCompletionProc)schedule_comp;           300
}

WDS SDS[3];
SDS[0].length = 0;
SDS[0].ptr = nil;
SDS[1].length = 0;
SDS[1].ptr = nil;
SDS[2].length = 0;

UDPiopb soundpb;                                                      310
soundpb.ioCRefNum = moviedata->myRefNum;
```

```
soundpb.udpStream = moviedata->udpStream;
soundpb.csCode = UDPWrite;
soundpb.csParam.send.reserved = 0;
soundpb.csParam.send.remoteHost = moviedata->addr;
soundpb.csParam.send.remotePort = moviedata->sPort;
soundpb.csParam.send.wdsPtr = (Ptr)SDS;
soundpb.csParam.send.checkSum = 0;
soundpb.csParam.send.sendLength = 0;
soundpb.csParam.send.userDataPtr = (Ptr)&thread;                    320
soundpb.ioCompletion = (UDPIOCompletionProc)schedule_comp;

UDPiopb sound[5];

for(i=0;i<5;i++){
sound[i].ioCRefNum = moviedata->myRefNum;
sound[i].udpStream = moviedata->udpStream;
sound[i].csCode = UDPWrite;
sound[i].csParam.send.reserved = 0;
sound[i].csParam.send.remoteHost = moviedata->addr;                 330
sound[i].csParam.send.remotePort = moviedata->vPort;
sound[i].csParam.send.wdsPtr = (Ptr)SDS;
sound[i].csParam.send.checkSum = 0;
sound[i].csParam.send.sendLength = 0;
sound[i].csParam.send.userDataPtr = (Ptr)&thread;
sound[i].ioCompletion = (UDPIOCompletionProc)schedule_comp;
}

TimeValue currentTime = 0,currentSTime = 0,nextDuration,actualTime;
TimerUPP mySyncProc = NewTimerProc(SyncProc);                       340
MyTask task;
task.task.tmAddr = mySyncProc;
task.task.tmWakeUp = 0;
task.task.tmReserved = 0;
task.A5world = SetCurrentA5();
InsXTime((QElemPtr)&task);
TimerUPP mySyncProc2 = NewTimerProc(SyncProc2);
```

```
MyTask task2;
task2.task.tmAddr = mySyncProc2;
task2.task.tmWakeUp = 0;                                                    350
task2.task.tmReserved = 0;
task2.A5world = SetCurrentA5();
InsXTime((QElemPtr)&task2);
long numSamples,SnumSamples,actualDPS;
Boolean firstcall = true;
Handle videoH = nil,soundH = nil;
long tempDuration,remainder=0,remainder2=0;
while (currentTime < endTime) {
        long size,Ssize;
        if (!vready) {                                                      360
                if (videoH) DisposeHandle(videoH);
                GetMediaSample(videoMedia,videoH = NewHandle(0),0,&size,
                                currentTime,&currentTime,&actualDPS,
                                nil,nil,1,&numSamples,nil);
                duration = -((actualDPS*1000000.0)+remainder)/videoTimeScale;
                remainder = actualDPS*1000000.0+remainder+
                                                videoTimeScale*duration;
                int bufNum = size/8192;
                int left = size-bufNum*8192;
                int num;                                                    370
                if (!left) num = bufNum; else num = bufNum+1;
                VDS[0].length = sizeof(long);
                VDS[0].ptr = (Ptr)&duration;
                VDS[1].length = sizeof(long);
                VDS[1].ptr = (Ptr)&num;
                VDS[2].length = 0;
                e = PBControlAsync((ParmBlkPtr)&videopb);
                thread.Suspend();
                char *p = (char *)*videoH;
                VDS[1].length = 0;                                          380
                for(int i=0;i<num-1;i++) {
                VDS[0].ptr = p;
                VDS[0].length = 8192;
```

```
                e = PBControlAsync((ParmBlkPtr)&videopb);
                thread.Suspend();
                p += 8192;
                }
                VDS[0].ptr = p;
                VDS[0].length = left;
                videopb.ioCompletion = (UDPIOCompletionProc)schedule_comp;
                e = PBControlAsync((ParmBlkPtr)&videopb);
                thread.Suspend();
                vready = true;
                currentTime += actualDPS*numSamples;
                }
        if (!sready) {
                if (soundH) DisposeHandle(soundH);
                e = GetMediaSample(soundMedia,soundH = NewHandle(0),0,
                                &Ssize, currentSTime,&currentSTime,
                                &actualDPS,nil,nil,4096,&SnumSamples,nil);
                char *p = (char *)*soundH;
                SDS[1].length = 0;
                SDS[0].ptr = p;
                SDS[0].length = Ssize;
                soundpb.ioCompletion = (UDPIOCompletionProc)schedule_comp;
                e = PBControlAsync((ParmBlkPtr)&soundpb):
                thread.Suspend();
                duration2 = -((actualDPS*SnumSamples*1000000.0)+
                                remainder2)/soundTimeScale;
                if (duration2 < -250000) duration2 += -250000;
                sready = true;
                if (firstcall) {
                        firstcall = false;PrimeTime((QElemPtr)&task,500);
                        PrimeTime((QElemPtr)&task2,0);
                        }
                currentSTime += actualDPS*SnumSamples;
                }
        }
duration = 0;
```

```
        VDS[0].length = sizeof(long);                                          420
        VDS[0].ptr = (Ptr)&duration;
        VDS[1].length = sizeof(long);
        VDS[1].ptr = (Ptr)&duration;
        VDS[2].length = 0;
        e = PBControlSync((ParmBlkPtr)&videopb);
        while (soundpb.ioResult || videopb.ioResult) LThread::Yield();
        if (soundH) DisposeHandle(soundH);
        if (videoH) DisposeHandle(videoH);
        DisposeMovie(schindler);
        CloseMovieFile(resRefNum);                                             430
        ready[me] = true;
        thread.Suspend();
        }
}
```

Bibliography

[1] D. P. Anderson. Metascheduling for continuous media. *ACM Transactions on Computer Systems*, 11(3), August 1993.

[2] D. P. Anderson and G. Homsy. A continuous media i/o server and its synchronization mechanism. *IEEE Computer*, 24(10), October 1991.

[3] D. C. A. Butlerman et al. A structure for transportable dynamic multimedia documents. In *Proc. 1991 Summer Usenix Conf.*, 1991.

[4] J. G. Hanko et al. Workstation support for time-critical appliations. In *Proceedings of the Second International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Heidelburg, November 1991.

[5] P. V. Rangan et al. A testbed for managing digital video and audio storage. In *Proc. 1991 Summer Usenix Conf.*, 1991.

[6] P. G. Milazzo. Shared video under unix. In *Proc. 1991 Summer Usenix Conf.*, 1991.